What is React Native?

React Native is an open-source framework developed by **Facebook** that enables developers to create mobile apps with **JavaScript** and **React**. In contrast to native development, which involves writing code separately for Android and iOS, React Native allows developers to write code once and execute it on both platforms. By leveraging native components and APIs, React Native fills the gap between **web development** and **mobile app development**, bringing the best of both worlds—efficiency and speed. Using React Native, developers can create **high-performance, cross-platform** applications that are native-looking and native-feeling.

Key Features of React Native

- **Cross-Platform Development:** Write once, run anywhere. React Native allows developers to create apps for both iOS and Android platforms using a shared codebase.
- **Native Performance:** React Native components are compiled into native code, which means that the apps deliver performance close to native apps.
- **Reusability:** React Native enables code reusability, which allows developers to use the same code for different platforms with minimal adjustments.
- **Hot Reloading:** This feature allows developers to instantly see the results of the latest change to the source code without rebuilding the app from scratch.
- **Component-Based Architecture:** Similar to React for the web, React Native uses a component-based architecture, making it easier to manage complex applications by breaking them down into smaller, reusable components.

Threads in React Native App

React Native uses multiple threads to manage the tasks involved in running an app. The main threads are:

UI Thread (Main Thread)

The **UI Thread** is responsible for rendering the user interface of the app. UI Thread, also known as **Main Thread**, is used for native android or iOS UI rendering. For example, In android this thread is used ensures that the app's interface is responsive and visually rendered.

JS Thread (JavaScript Thread)

JS thread or Javascript thread is the thread where the logic will run. For e.g., this is the thread where the application's javascript code is executed, api calls are made, touch events are processed and many other. For performance, **React Native** ensures that the JS Thread sends updates to the UI Thread before the next frame rendering deadline (which is around 16.67ms for 60 frames per second on iOS).

- If the JS Thread performs complex computations that take **more than 16.67ms**, the UI can become sluggish or unresponsive. This is why the JS Thread must be optimized to avoid long delays in execution.
- Notably, some components like **NavigatorIOS** and **ScrollView** run completely on the UI Thread, which prevents them from being blocked by slow JS thread operations.

Native Modules Thread

The **Native Modules Thread** comes into play when the app needs to access platform-specific APIs (e.g., **camera, GPS)**. This thread allows React Native to call native functions from Android or iOS directly, bridging the gap between JavaScript and native code.

Render Thread (Android Only)

In Android (specifically for version L and later), the **Render Thread** is responsible for generating OpenGL commands used to draw the UI. The Render Thread enables the app to efficiently render graphics and images on the screen.

Process Involved in Working of React Native

React Native works by following a sequence of processes from app startup to rendering:

1) App Startup

On app startup, the main thread is used for executing and loading the JavaScript bundles. The JS bundles have the app's logic and UI components.

2) JavaScript Execution

After the successful loading of the JavaScript bundle, the **JS Thread** takes over the execution. This enables the JS Thread to execute the heavy computations without influencing the UI Thread, which keeps the user interface responsive.

3) Virtual DOM and Reconciliation

React Native employs Reconciliation to effectively render UI updates. The Reconciler then contrasts the current virtual DOM with the new virtual DOM ("**diffing**"), and where there are updates to be made, sends them off to the **Shadow Thread.**

4) Layout Calculation

The Shadow Thread determines the layout of the UI components and passes the layout parameters to the UI Thread. "**Shadow**" in this case is the virtual UI that is created during this calculation. The layout consists of position, size, and other characteristics of UI components.

5) Screen Rendering

As only the UI Thread is allowed to display UI on the screen, the layout data computed by the **Shadow Thread** is dispatched to the UI Thread. The UI Thread renders the final result on the screen and displays it to the user.

Parts of React Native

React Native can broadly be divided into three primary parts:

React Native - Native Side

The native side comprises the native modules and views that belong to the **Android** or **iOS** platform. It contains features such as the native UI and platform-specific APIs.

React Native - JS Side

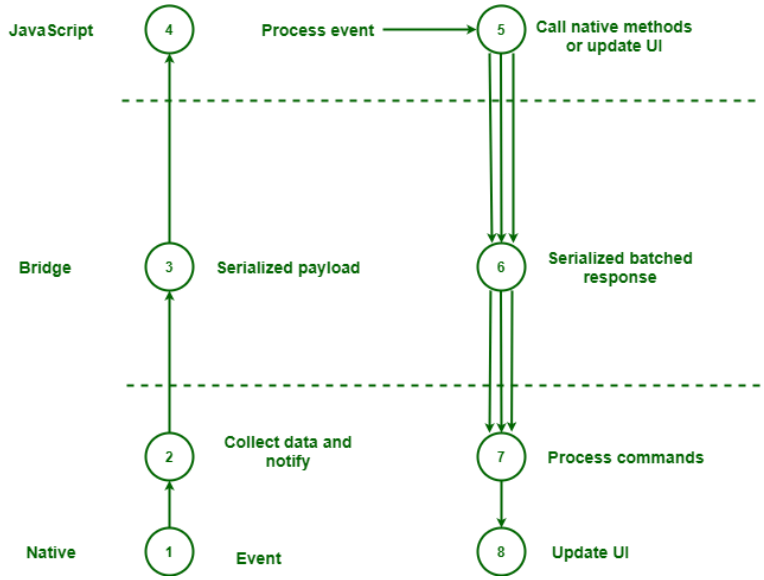
The JS side is where the JavaScript code executes. This is the code you, as a developer, write using React

and JavaScript. It handles app logic, state, events, and UI rendering.

React Native - Bridge

The Bridge provides a communication mechanism between the JS side and native side. Asynchronous communication is supported, meaning that JavaScript and native code will not block either thread. It enables sending updates, data, and events from JavaScript to native modules.

Architecture: Following is the diagram that explains the structure of React Native architecture :



React Native Architecture

Three layers are stated in the diagram above:

1. **JavaScript**
2. **Bridge**
3. **Native**

The Bridge in React Native is the layer that permits the JavaScript and the Native modules to interact with each other and is primarily a carrier layer that conducts nonparallel chronological grouped feedback messages from JavaScript to Native modules. In the picture given above the Native layer is shown last since this layer is nearest to the device itself.

When an incident is implemented on the Native layer then it might be a timer, touch, or network request, primarily, any incident associating device Native modules. Its data is assembled and this data is sent to the React Native Bridge as a serialized message. After that, the Bridge proceeds this message towards the JavaScript layer.

The JavaScript layer is an incident loop. After the React Native Bridge proceeds the Serialized payload to JavaScript then the incident is processed and the application logic becomes active.

Threading Model: Now, after all the things that we have discussed above it is significant to recognize that all the things discussed above are done on three major threads:

1. UI (the application's main thread)

2. Native modules
3. JavaScript runtime

Let's understand about each of the above threads:

The UI (the application's main thread): It is the Native thread where the native-level perception takes place and It is the place where our platform of choices such as iOS or Android, carries out drawing, styling, and measuring.

Native modules: If the application accesses any Native APIs then the accessing is done on a different Native modules thread. For instance, if the camera, location, photos, and any further Native API is accessed then generally arrangement and indications are also concluded on this thread.

JavaScript runtime: It is the thread where every JavaScript application code will run. After all, it is based on a JavaScript incident loop, so it is steadier than the UI thread. Hence, when complicated calculations are done in an application that conducts many UI changes then these may give on to poor performance.

JavaScript

React Native uses Babel as a JavaScript compiler to allow us to develop in the latest version of JavaScript, ES2016.

Starting the project Create React Native App To begin, we're going to use Create React Native App (CRNA), a tool that makes it extremely easy to get started with React Native. If you've used Create React App before, you'll notice similarities here in that no build configuration is required to get up and running. We can install it globally using yarn.

yarn

yarn16 is a node package manager that automates the process of managing all the required dependencies and packages from npm, an online repository of published JavaScript libraries and projects, in an application. This is done by defining all our dependencies in a single package.json file.

Watchman

Watchman is a file watching service that watches files and triggers actions when they are modified. If you use macOS as your operating system, the Expo and React Native documentation recommend installing Watchman for better performance.

Expo

Expo is a platform that provides a number of different tools to build fully functional React Native applications without having to write native code. Beginning a project with CRNA automatically creates an application that leverages Expo's development environment. A benefit of leveraging Expo is that building an application does not require using Xcode for iOS, or Android Studio for Android. This means that developers can build native iOS applications without even owning a Mac computer. Using CRNA and Expo is the easiest way to get started with React Native and is recommended in the React Native documentation

Components

With newer versions of JavaScript, we can define objects with properties using classes. React Native lets us use this syntax to create components. Let's take a look at a visual breakdown of the components in

our application:

We have an App component that represents the entire screen and contains the weather information displayed to the use. Inside of this component, we have a SearchInput component that allows us to search for different cities.

App is the first component created with a default CRNA application. Let's take a look at its file:

weather/1/App.js

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
        <Text>Changes you make will automatically reload.</Text>
        <Text>Shake your phone to open the developer menu.</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Notice how we have a class defined in our file named App that extends React.Component. Using extends allows us to declare a class as a subclass of another class. In here, we've defined App as a subclass of React.Component. This is how we specify a specific class to be a component in our application.

We can also attach methods as properties to classes, and the same applies to component classes in React Native. We can see we already have one for this component, the render method:

weather/1/App.js

```
  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
        <Text>Changes you make will automatically reload.</Text>
        <Text>Shake your phone to open the developer menu.</Text>
      </View>
    );
  }
```

What we see on our device when launching our device matches what we see described in this method. The render() method is the only required method for a React Native component. React Native uses the return value from this method to determine what to render for the component. When we use React Native, we represent different parts of our application as components. This means we can build our app using different reusable pieces of logic with each piece displaying a specific part of our UI.

Let's break down what we already have in terms of components:

• Our entire application is rendered with App as our top-level component. Although created automatically as part of setting up a new CRNA project, this component is a custom component responsible for rendering what we need in our application.

• The View component is used as a layout container.

• Within View, we use the Text component to display lines of text in our application. Unlike App, both View and Text are built-in React Native components that are imported and used in our custom component. We can see that our App component uses and returns an HTML-like structure. This is JSX, which is an extension of JavaScript that allows us to use an XML-like syntax to define our UI structure.

JSX

When we build an application with React Native, components ultimately render native views which are displayed on our device. As such, the render() method of a component needs to describe how the view should be represented. In other words, React Native allows us to describe a component's iOS and Android representation in JavaScript. JSX was created to make the JavaScript representation of components easier to understand. It allows us to structure components and show their hierarchy visually in markup. Consider this JSX snippet:

```
<View>
  <Text style={{ color: 'red' }}>
    Hello, friend! I am a basic React Native component.
  </Text>
</View>
```

In here, we've nested a Text component within a View component. Notice how we use braces ({}) around an object ({ color: 'red' }) to set the style property value for Text. In JSX, braces are a delimiter, signaling to JSX that what resides in-between the braces is a JavaScript expression. The other delimiter is using quotes for strings, like this: Even though the JSX above might look similar to HTML, it is actually just compiled into JavaScript function calls (ex: React.createElement(View)). For this reason, we need to

import React at the top of any file that contains JSX.

During runtime React Native takes care of rendering the actual native UI for each component.

ReactJS State vs Props

In React, State allows components to manage and update internal data dynamically, while Props enables data to be passed from a parent component to a child component. Understanding their differences and use cases is essential for developing efficient React applications.

State in React

State is a built-in object in React components that holds data or information about the component. It is mutable, which means it can be updated within the component using the setState method in class components or the useState hook in functional components.

- State is local to the component and cannot be accessed by child components unless passed down as props.
- It is mutable, meaning it can change over time based on user interactions or API responses.
- When state updates, the component re-renders to reflect the changes.
- Managed using useState in functional components or this.setState in class components.

useState: The useState hook is used to declare state variables in functional components. It allows us to read and update the state within the component.

Syntax

const [state, setState] = useState(initialState);

- **state:** The current value of the state.
- **setState:** A function used to update the state.
- **initialState:** The initial value of the state, which can be a primitive type or an object/array

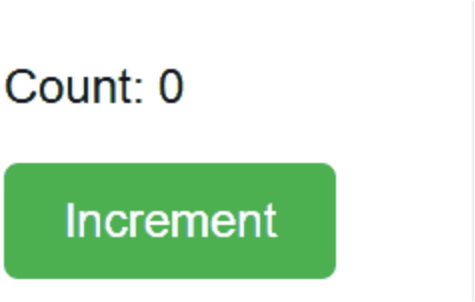
---

```
import React, { useState } from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';

export default function Counter() {
  const [count, setCount] = useState(0);

  return (
    <View style={styles.container}>
      <Text style={styles.text}>Count: {count}</Text>
      <Button title="Increment" onPress={() => setCount(count + 1)} />
    </View>
  );
}
```

Output



In this example

- useState(0) initializes the state variable count with the value 0.
- The setCount function is used to update the state whenever the button is clicked. This triggers a re-render, updating the displayed count.

Props

Props (short for Properties) are used to pass data from a parent component to a child component. Unlike state, props are immutable, meaning they cannot be modified within the receiving component.

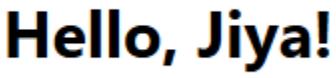
- Props allow components to be reusable and dynamic.
- Props are read-only and cannot be changed by the child component.
- They help in data communication between components.
- Passed as attributes in JSX elements.

```
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

function Greeting({ name }) {
  return <Text style={styles.text}>Hello, {name}!</Text>;
}

export default function App() {
  return (
    <View style={styles.container}>
      <Greeting name="Jiya" />
    </View>
  );
}
```

Output



Here, the Greeting component receives a prop named name and displays a personalized message. The App component passes the value "Jiya" as a prop.

Difference between State and Props in React

Aspect	State	Props
Modification	Can be changed by the component itself.	Cannot be changed by the receiving component; props are read-only.
Communication	Facilitates communication within a component.	Facilitates communication between components (from parent to child).
Re-rendering	Changes in state trigger a re-render of the component where the state is modified.	Changes in props cause a re-render of the child component that receives them, but the parent manages them.
Component Type	State is used in **class components** (via this.state and this.setState) or **functional components** (via useState).	Props are used in both class and functional components.
Effect on Parent	Changing state only affects the component where the state is defined.	Changing props doesn't affect the parent component directly; the parent controls the props passed to the child.
Responsibility	Managed by the component itself.	Managed by the parent component.
Examples	const [count, setCount] = useState(0); (functional component with hook)	<Button color="blue" onClick={handleClick} /> (passing props to a child component)

When to Use State and Props?

- **Use State** when you need to manage data that can change over time within a component (e.g., form inputs, counters, UI toggles).

- **Use Props** when you need to pass data from a parent component to a child component to make components reusable.

React Native is a powerful framework through which developers can create mobile apps with the help of JavaScript. By utilizing several threads (**UI, JS, Native Modules, and Render Thread**) and streamlined processes for rendering, React Native achieves high performance and responsiveness. Knowing the React Native flow—from loading the JavaScript bundle to the rendering of UI updates—will make you write more efficient code and improve your app's performance.

By dividing React Native into three components—**native side, JS side, and the bridge**—this architecture enables smooth interaction between native code and JavaScript code, offering the best of both worlds.

Native Components / Core Components of React Native

Native components in the context of React Native refer to UI elements that are rendered using native platform-specific views. These components allow developers to build user interfaces that look and feel like native apps on iOS and Android. React Native bridges JavaScript code with the native platform's components to provide a seamless and performant user experience.

Some examples of native components in React Native include:

1. **View:** The **<View>** component is similar to a ** ** in web development. It is a container used to group and layout other components and does not render any visual output itself.

```
import React from 'react';
import { StyleSheet, View, Text } from 'react-native';

function App() {
  return (
    <View style={styles.container}>
      <Text>Hello, World!</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fffff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});

export default App;
```

2. **Text:** The **<Text>** component is used for displaying text content. It renders text in a platform-specific manner, taking into account the platform's fonts and text rendering features. While it has some basic styling properties, you usually nest it inside other components (e.g., View) to create more complex UIs.

```
import React from 'react';
import { Text, StyleSheet } from 'react-native';
const SimpleText = () => {
  return (
    <Text style={styles.text} numberOfLines={2} onPress={() => alert('Hello')}>
      This is an example of a Text component in React Native. Tap on me!
    </Text>
  );
};
const styles = StyleSheet.create({
  text: {
    fontSize: 16,
    color: 'blue',
    textAlign: 'center',
    margin: 10,
    fontFamily: 'Arial',
  },
});
export default SimpleText;
```

3. **Image:** The **<Image>** component is used for displaying images. It loads and displays images efficiently using the native platform's image handling capabilities. The Image component is used to display images in a React Native application. It allows you to load and display local as well as remote images, providing essential props and methods for better image handling and customization.
   To use the Image component, you need to import it from 'react-native':

```
import { Image } from 'react-native';
```

To display a local image in the application, you have to require it in the source prop of the Image component. Place the image file in your project directory and use the following syntax:

```
<Image source={require('./path/to/your/image.png')} />
```

To display a remote image from a URL, you need to set the source prop with a url object:

```
<Image source={{ uri: 'https://path/to/your/remote/image.png' }} />
```

Keep in mind that you need to define the dimensions (width and height) when using remote images:

```
<Image source={{ uri: 'https://path/to/remote/image.png' }} style={{ width: 200, height: 200 }} />
```

4. **ScrollView:** In React Native, the ScrollView is a generic scrolling container used to provide a scrollable view to its child components. It is useful when you need to display scrollable content larger than the screen, such as lists, images, or text. A ScrollView must have a bounded height in order to properly work.

   Here's a simple example of how to use the ScrollView component in your React Native app:

```
import React from 'react';
import { ScrollView, Text } from 'react-native';

const MyScrollView = () => {
  return (
    <ScrollView>
      <Text>Item 1</Text>
      <Text>Item 2</Text>
      <Text>Item 3</Text>
      <Text>Item 4</Text>
      <Text>Item 5</Text>
      <Text>Item 6</Text>
    </ScrollView>
  );
}

export default MyScrollView;
```

5. **TextInput**: TextInput is a core component in React Native that allows the user to enter text. It is commonly used to collect user data, like emails or passwords. You can customize the appearance of TextInput by using various props such as placeholder, multiline, maxLength, and more.

   Here's a basic example of using TextInput:

```
import React, { useState } from 'react';
```

```
import { TextInput, View, Button } from 'react-native';
const App = () => {
  const [text, setText] = useState('');
  const handleSubmit = () => {
    alert('You entered: ' + text);
  };
  return (
    <View>
      <TextInput
        style={{ height: 40, borderColor: 'gray', borderWidth: 1 }}
        onChangeText={text => setText(text)}
        value={text}
        placeholder="Enter text here"
      />
      <Button
        onPress={handleSubmit}
        title="Submit"
      />
    </View>
  );
};
```

---

6. Button: A Button is a built-in React Native component used to create clickable buttons. It is a simple, customizable and easy-to-use component that captures touches and triggers an onPress event when pressed.

   Here's a simple example of how to create a Button in React Native:

```
import React from 'react';
import { Button } from 'react-native';

const MyButton = () => {
  const onPressHandler = () => {
    alert('Button Pressed');
  };

  return (
    <Button
      title="Click me"
      color="#841584"
      onPress={onPressHandler}
    />
  );
```

```
};

export default MyButton;
```

---

7. StatusBar: The StatusBar component is used to control the appearance of the status bar on the top of the screen. It may strike as a bit unusual since, unlike other React Native components, it doesn't render any visible content. Instead, it sets some native properties that can help customize the look of status bars on Android, iOS, or other platforms.

   To use the StatusBar component, you need to import it from 'react-native' and use it in your React component. Here's an example:

```
import React from 'react';
import { View, StatusBar } from 'react-native';
const App = () => {
  return (
    <View>
      <StatusBar barStyle="dark-content" backgroundColor="#F0F0F0" />
      {/* Your other components */}
    </View>
  );
};
export default App;
```

---

8. **Touchable Components:** React Native provides various touchable components like **<TouchableOpacity>, <TouchableHighlight>,** and **<Touchab1eWithoutFeedback>,** allowing developers to add touch interaction to elements. n React Native, Touchable components are used to handle user interactions like taps, long presses, and double-taps on the appropriate elements.

   TouchableOpacity is The opacity of the wrapped view is decreased when it's active.

```
import { TouchableOpacity } from 'react-native';

<TouchableOpacity
  onPress={() => {
    alert('Tapped!');
  }}>
  <Text>Tap me</Text>
```

</TouchableOpacity>

React Native UI Component	Android View	iOS View	Web Analog	Description
<View>	<ViewGroup>	<UIView>	A non-scrolling	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
<Text>	<TextView>	<UITextView>	<p>	Displays, styles, and nests strings of text and even handles touch events
<Image>	<ImageView>	<UIImageView>	<img>	Displays different types of images
<ScrollView>	<ScrollView>	<UIScrollView>		A generic scrolling container that can contain multiple components and views
<TextInput>	<EditText>	<UITextField>	<input type="text">	Allows the user to enter text

The key advantage of using native components is that they provide a more authentic native look and feel, as they leverage the native platform's UI elements. React Native bridges the gap between JavaScript and the native platform, ensuring that the app's UI components are rendered natively; resulting in better performance and user expense.

By using these native components, React Native developers can build apps that are visually consistent with native apps on both iOS and Android, making the development process efficient and enabling code reuse across different platforms.

STATES – COUNTER APP

```
import React, { useState } from 'react';
```

```
import { View, Text, Button, StyleSheet } from 'react-native';

export default function Counter() {
  const [count, setCount] = useState(0);

  return (
    <View style={styles.container}>
      <Text style={styles.text}>Count: {count}</Text>
      <Button title="Increment" onPress={() => setCount(count + 1)} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#fff',
  },
  text: {
    fontSize: 24,
    marginBottom: 20,
  },
});
```

PROP – GREETING APP

```
import React, { useState } from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';

export default function Counter() {
  const [count, setCount] = useState(0);

  return (
    <View style={styles.container}>
      <Text style={styles.text}>Count: {count}</Text>
      <Button title="Increment" onPress={() => setCount(count + 1)} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
```

```
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#fff',
  },
  text: {
    fontSize: 24,
    marginBottom: 20,
  },
});
```

COMMANDS

CREATE APP	npx create-expo-app myapp --template blank
START APP	npx expo start
To open in vs code	code .
To create directory	md directoryname
To navigate to directory	cd directoryname
to delete directory	rmdir /s directoryname
To list contents	dir
To move out of directory	cd..
To check connection To show list of devices	adb devices emulator -list-avds
To restart app	cd android gradlew clean cd .. npx react-native run-android
To run app on avd	npx react-native run-android

useState: The useState hook is used to declare state variables in functional components. It allows us to read and update the state within the component.

Syntax

const [state, setState] = useState(initialState);

- **state:** The current value of the state.
- **setState:** A function used to update the state.
- **initialState:** The initial value of the state, which can be a primitive type or an object/array